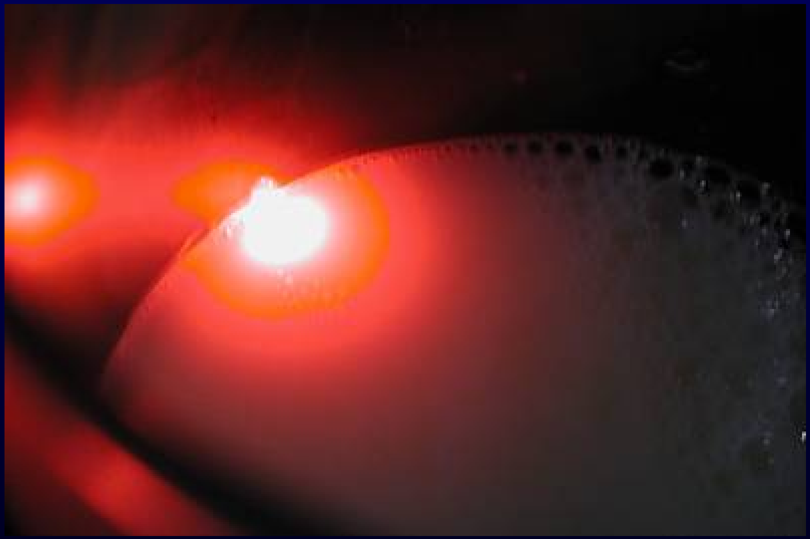
Subsurface Scattering Using Point Clouds

Mario Marengo

What Is It?

- SSS is a GLOBAL effect
- Can be split into two separate components: Single and Multiple scattering.
- Single scattering is the directional component, and Multiple scattering the diffuse component.
- The split is arbitrary and has to do with implementation, not light behaviour.

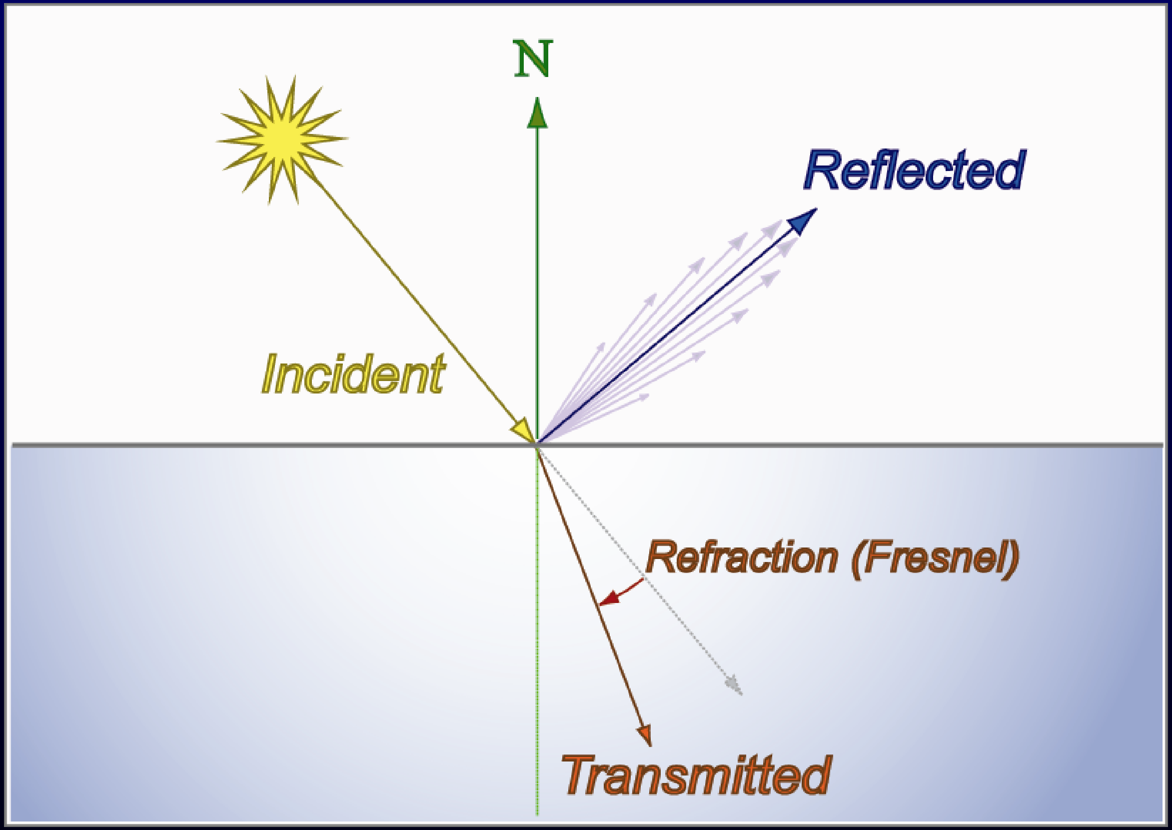


Multiple Scattering

Using Pixar’s approach from the Siggraph 2003 Renderman notes: “Human Skin For Finding Nemo”.

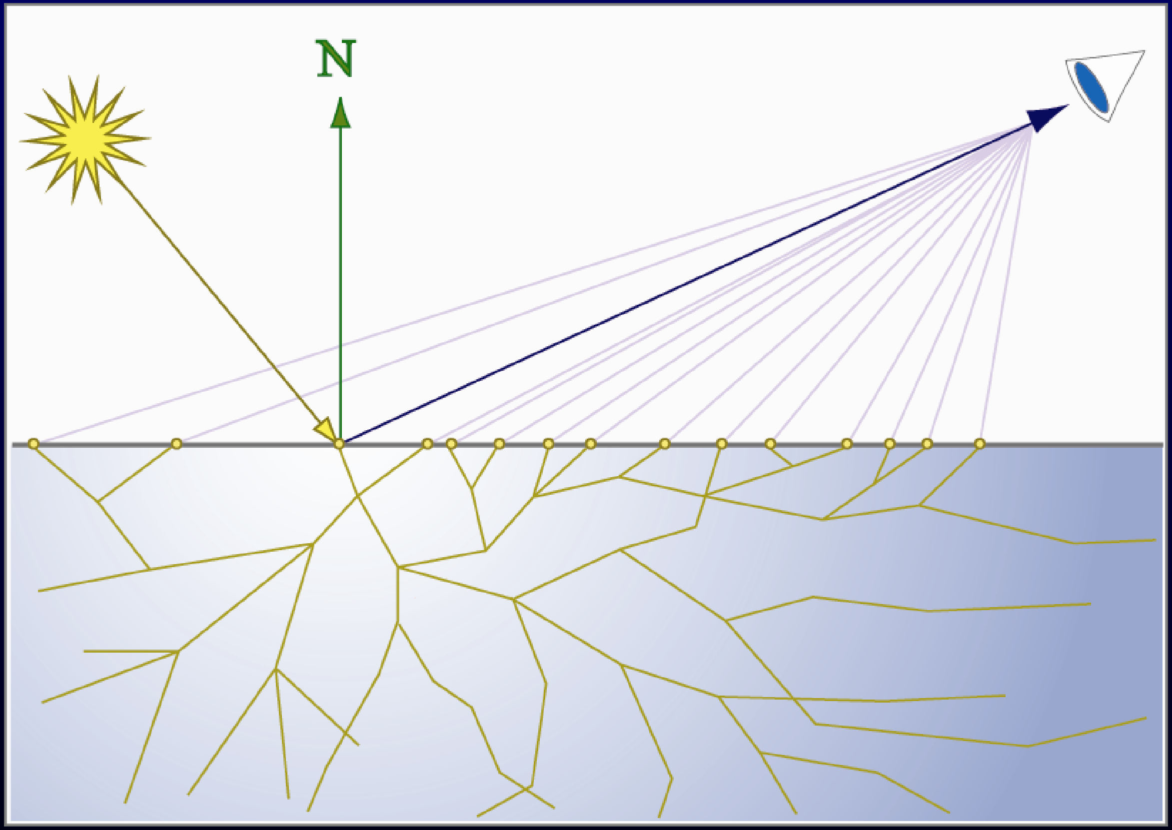
Light Path: Arrival

- A portion of the incident light is reflected, and the remaining amount transmitted.
- Fresnel functions calculate these factors.
- Local illumination models deal with the reflected portion and ignore transmission/absorption. SSS takes them it into account.



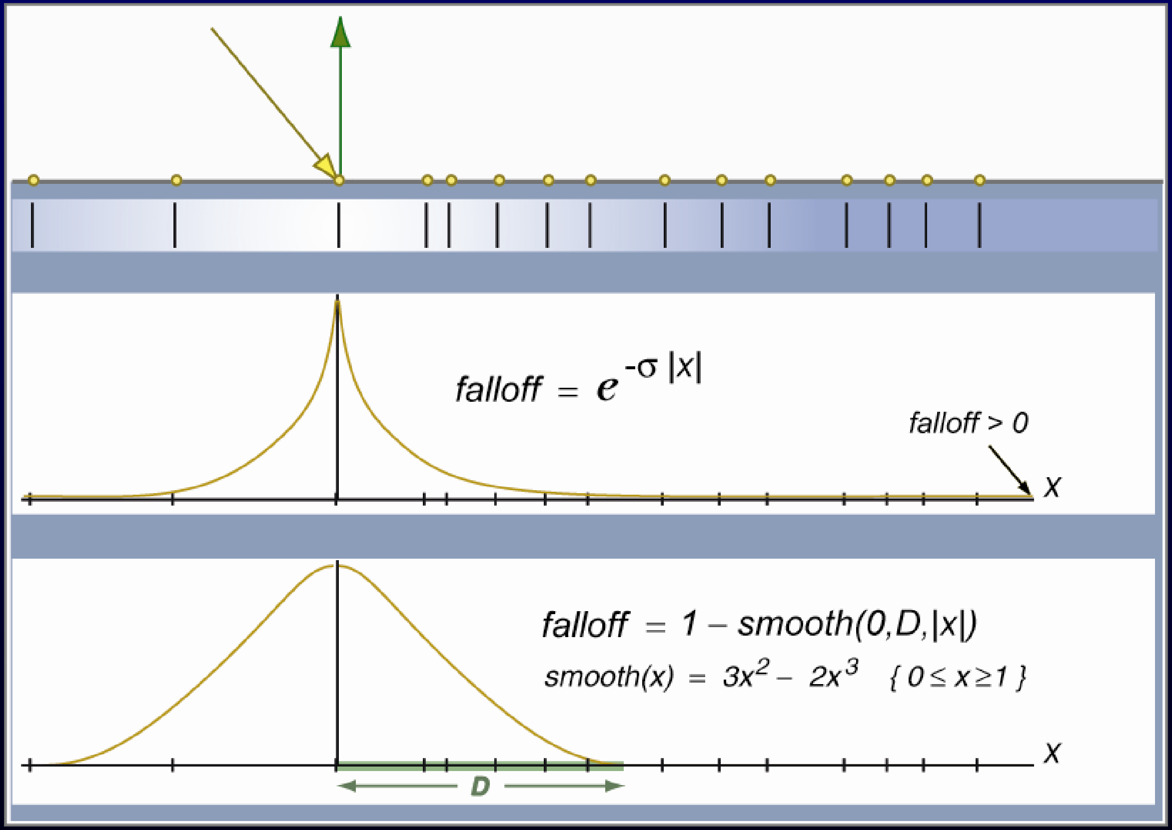
Light Path: Scatter

- Transmitted light bounces (scatters) inside the medium.
- Parts of it may exit the object at locations very distant from the point of incidence.
- As it travels it gradually looses energy until it becomes completely extinguished.



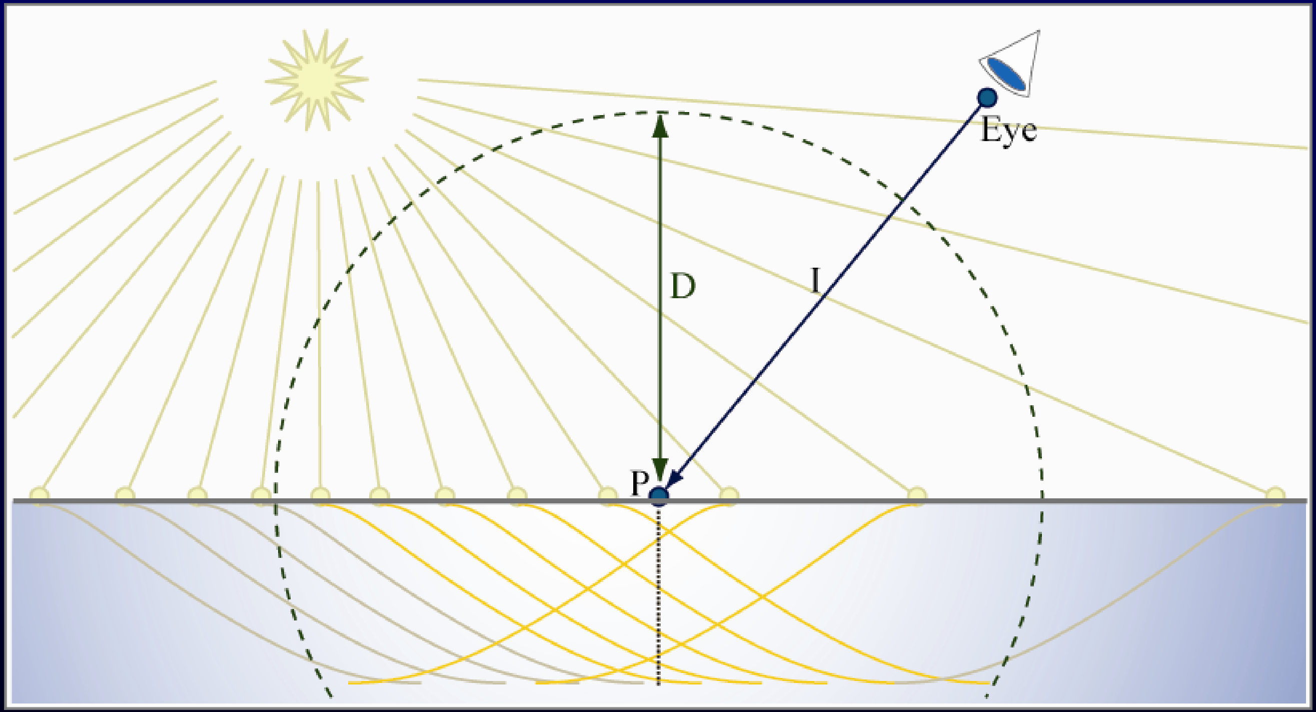
Extinction

- Physically correct model uses exponential curve.
- The Pixar model uses RSL's smoothstep() function, which is the same as VEX's smooth().
- Smooth() allows us to force intensity to reach zero at some user-defined distance D.
- We ignore the boundary!



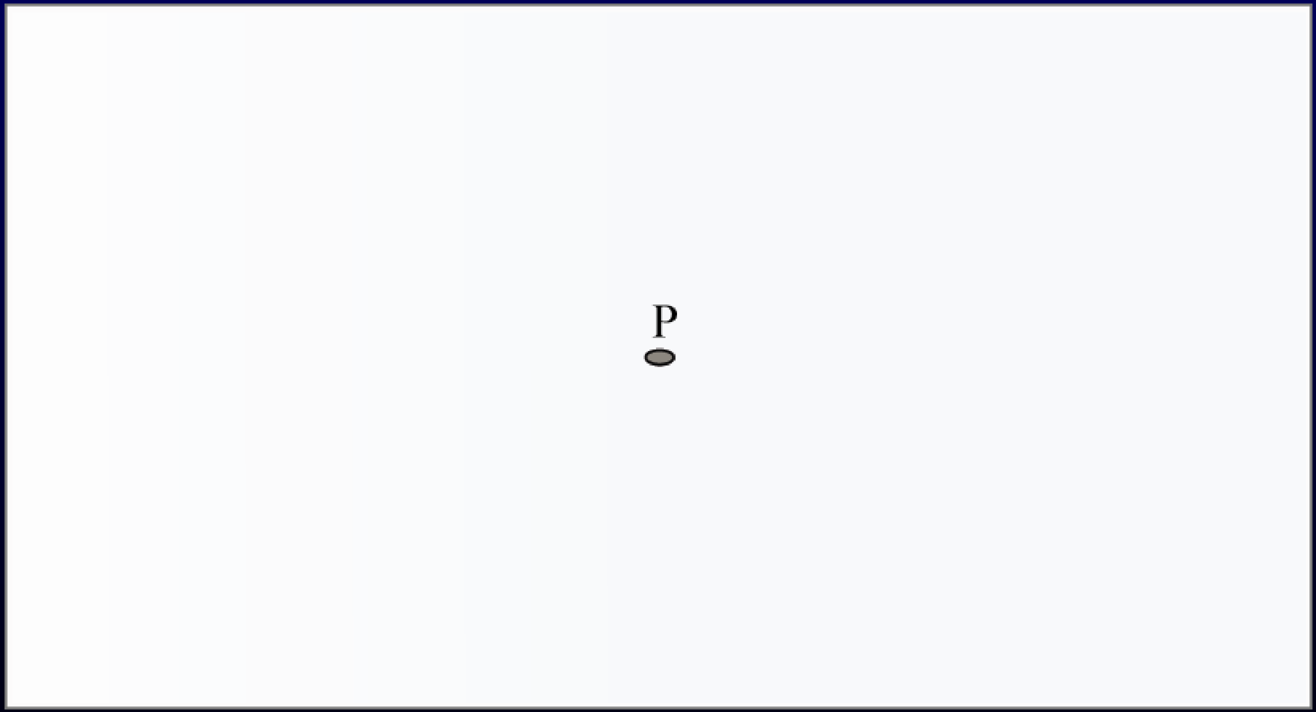
Extinction Viewed From The Shader

- Given a surface position P and a user-defined scattering distance D.
- We must sample the amount of light arriving within a distance D from P and apply our extinction curve to each sample.
- No way to sample a large neighbourhood around P inside a shader.
- Solution: Point Clouds! -- let the neighbouring points be the points in a point cloud.



Sampling Strategy

1. Start with a shading position P…

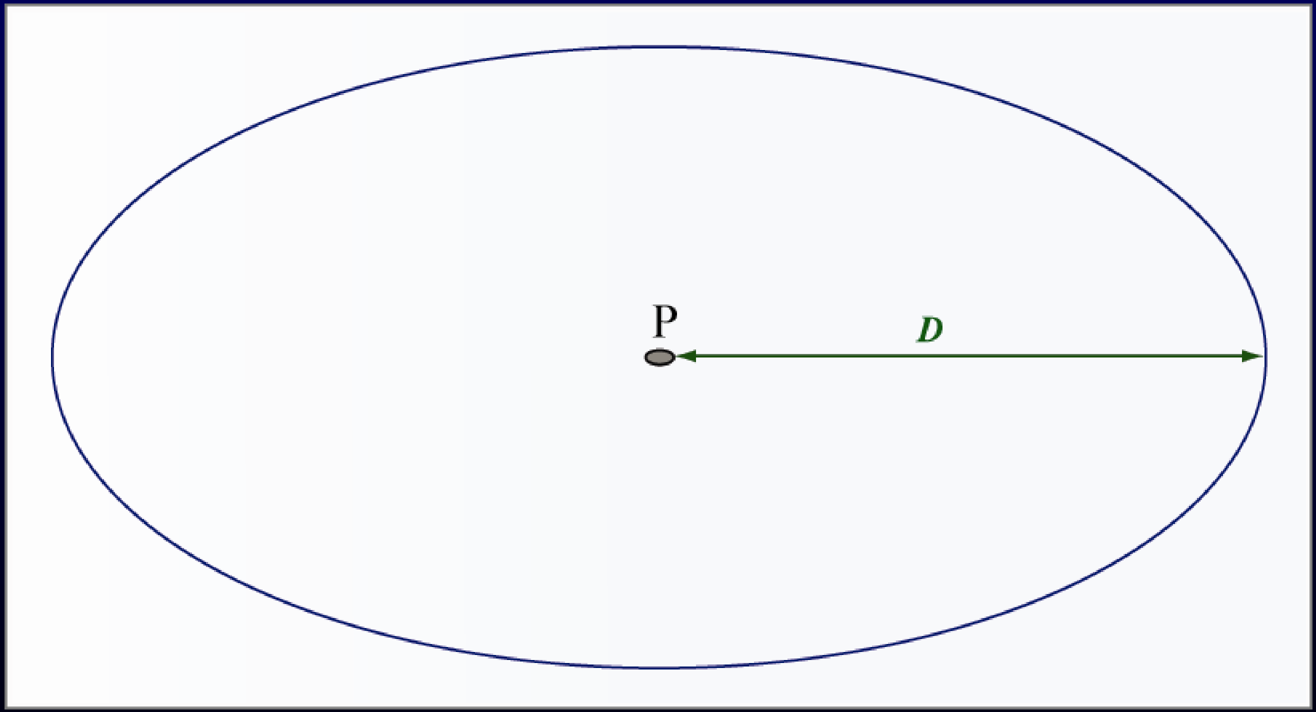


Sampling Strategy

1. Start with a shading position ***P*** and a user-provided "Scattering Distance" ***D***.

Sampling Strategy

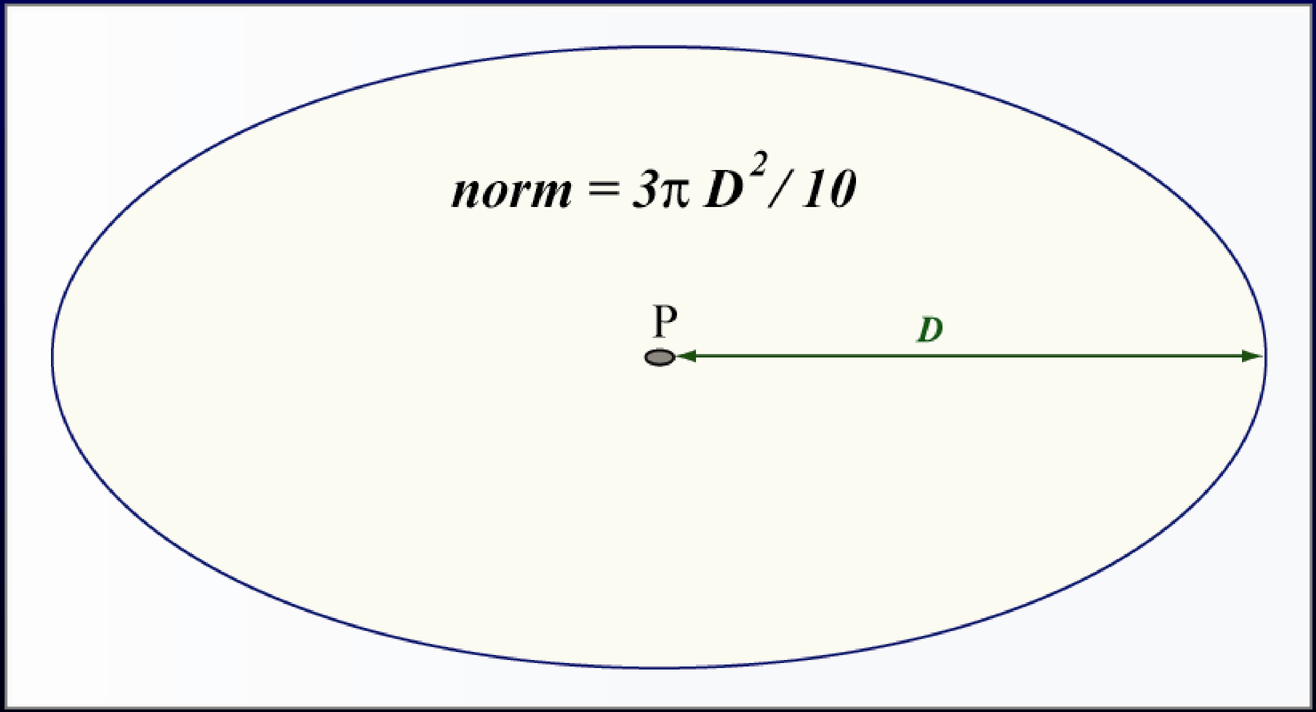
1. Start with a shading position ***P*** and a user-provided "Scattering Distance" ***D***.
   - We know that only points within a radius ***D*** of ***P*** can possibly contribute to SSS.



Sampling Strategy

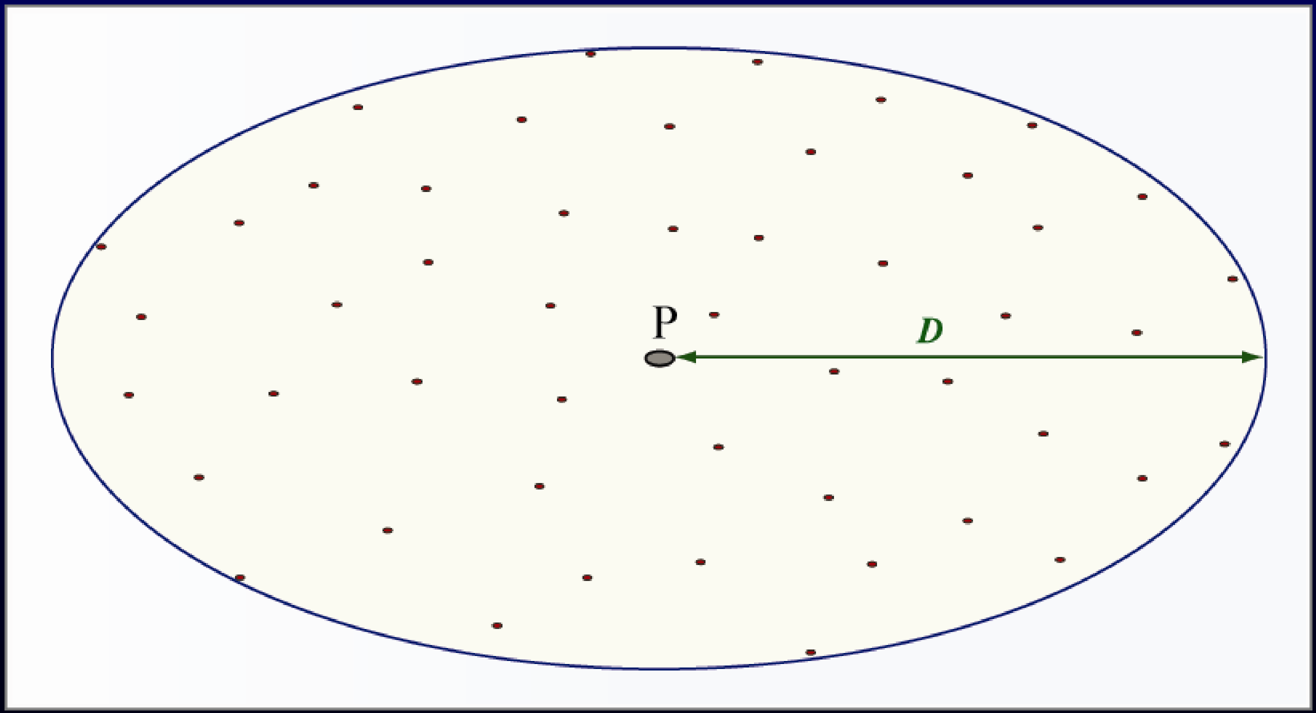
1. Start with a shading position P and a user-provided "Scattering Distance" D.
   - We know that only points within a radius D of P can possibly contribute to SSS.
2. Calculate a normalizing factor for this radius: $norm = 3\pi D^2 / 10$.

   For an exponential function of the form $e^{-\sigma r/D}$ *this factor would be*
   $2 D^2 e^{-\sigma} \pi (e^{\sigma} - \sigma - 1) / \sigma^2$



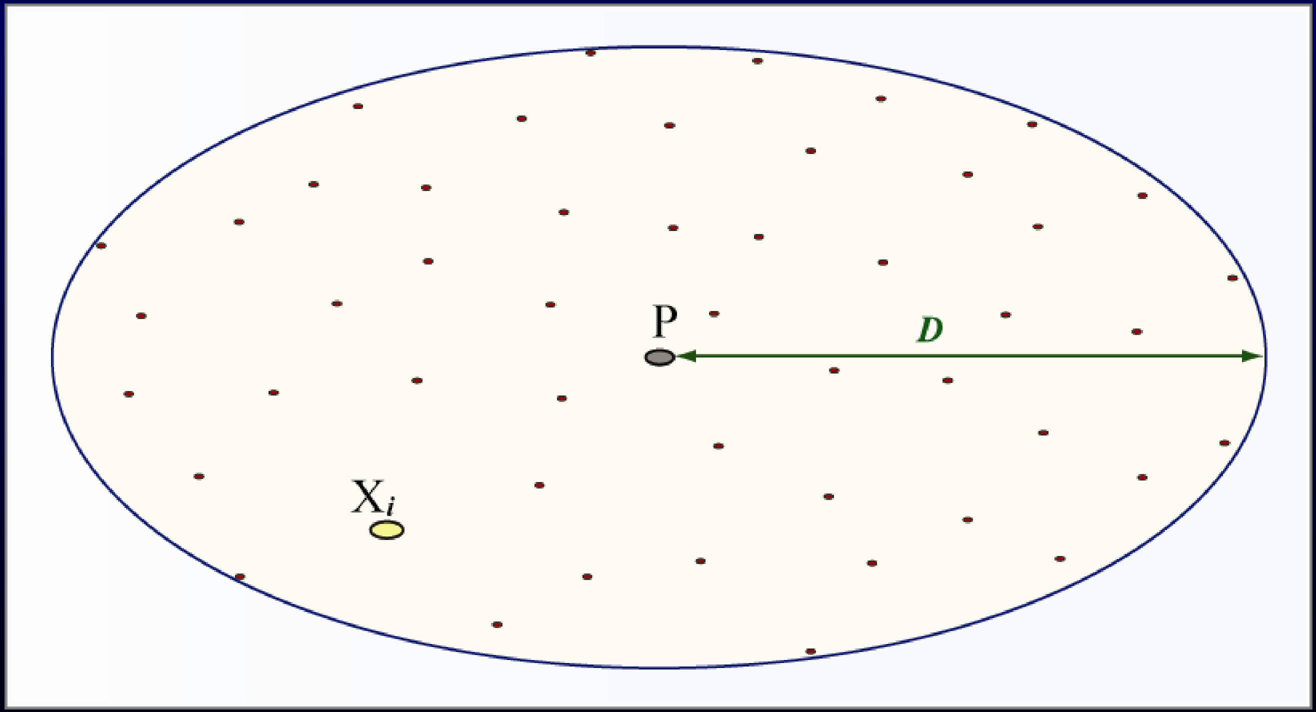
Sampling Strategy

1. Start with a shading position P and a user-provided "Scattering Distance" D.
   - We know that only points within a radius D of P can possibly contribute to SSS.
2. Calculate a normalizing factor for this radius: $norm = 3\pi D^2 / 10$.
3. Distribute sample points X_i within a radius D of shade point P.



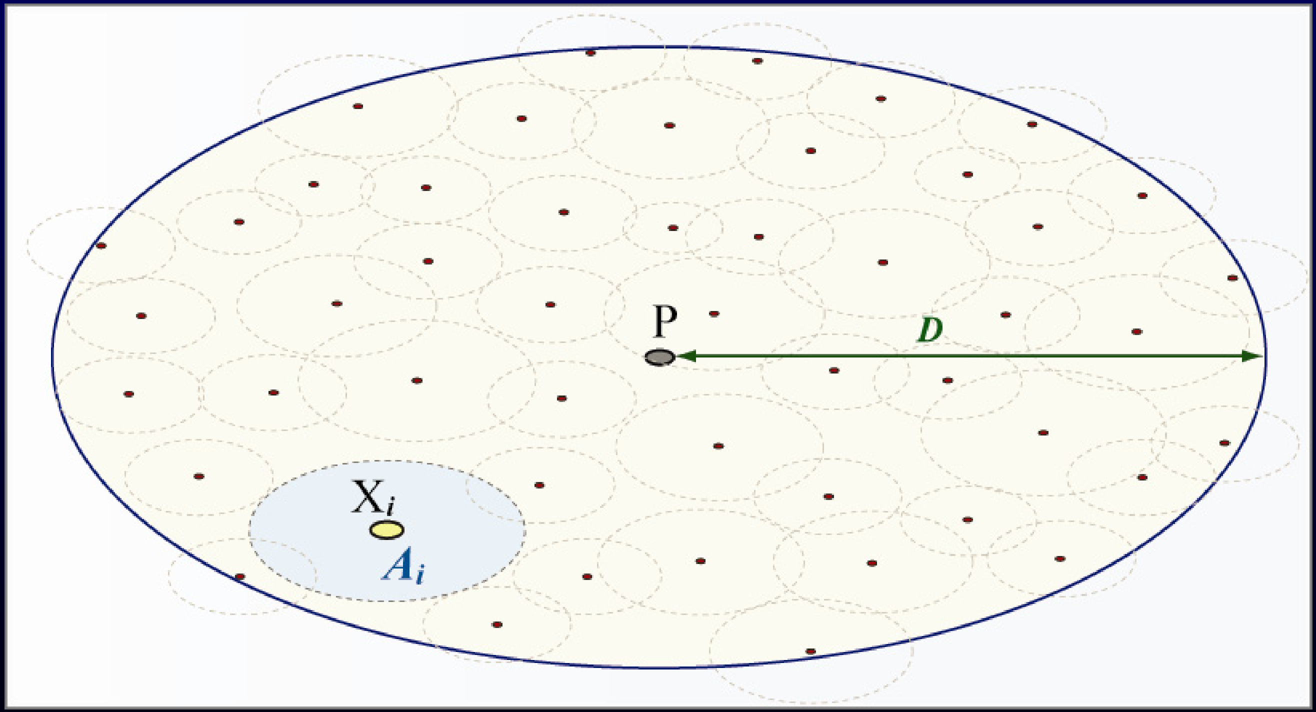
Sampling Strategy

1. Start with a shading position P and a user-provided "Scattering Distance" D.
   - We know that only points within a radius D of P can possibly contribute to SSS.
2. Calculate a normalizing factor for this radius: $norm = 3 \pi D^2 / 10$.
3. Distribute sample points X_i within a radius D of shade point P.
   - For each sample point X_i:



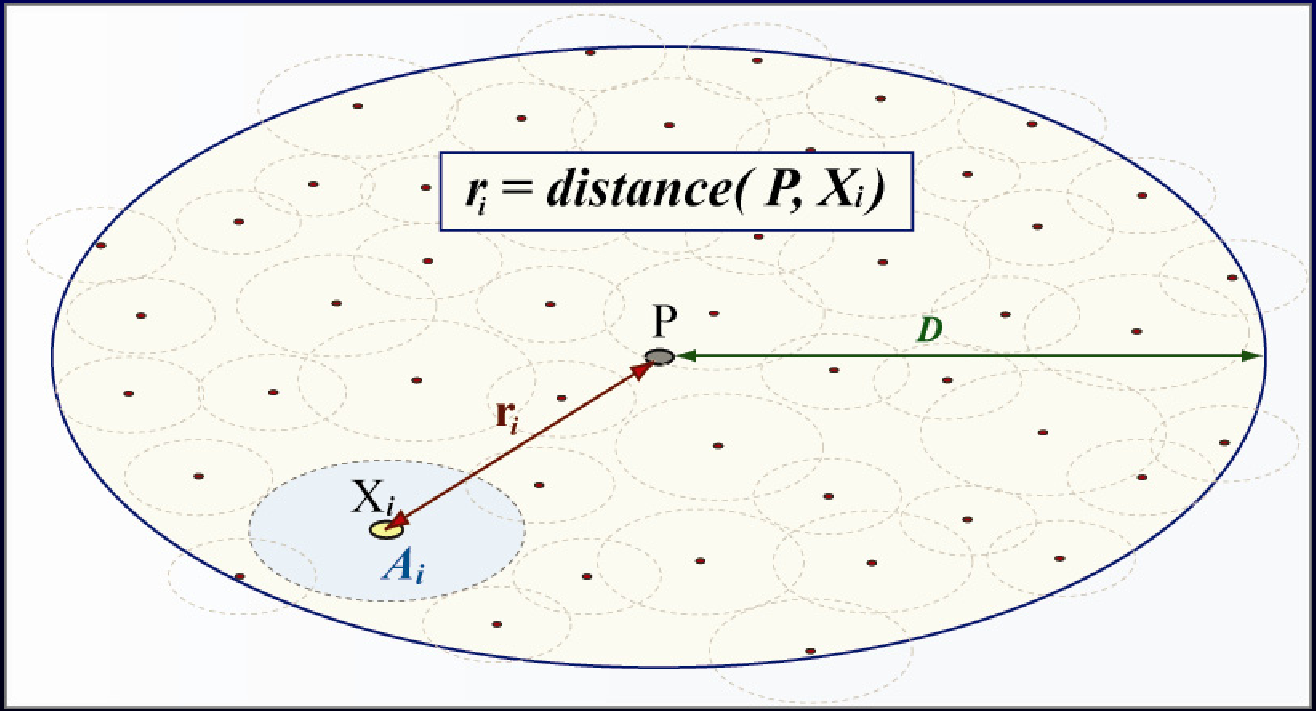
Sampling Strategy

1. Start with a shading position P and a user-provided "Scattering Distance" D.
   - We know that only points within a radius D of P can possibly contribute to SSS.
2. Calculate a normalizing factor for this radius: $norm = 3 \pi D^2 / 10$.
3. Distribute sample points X_i within a radius D of shade point P.
   - For each sample point X_i:
     1. Calculate representative surface area A_i.



Sampling Strategy

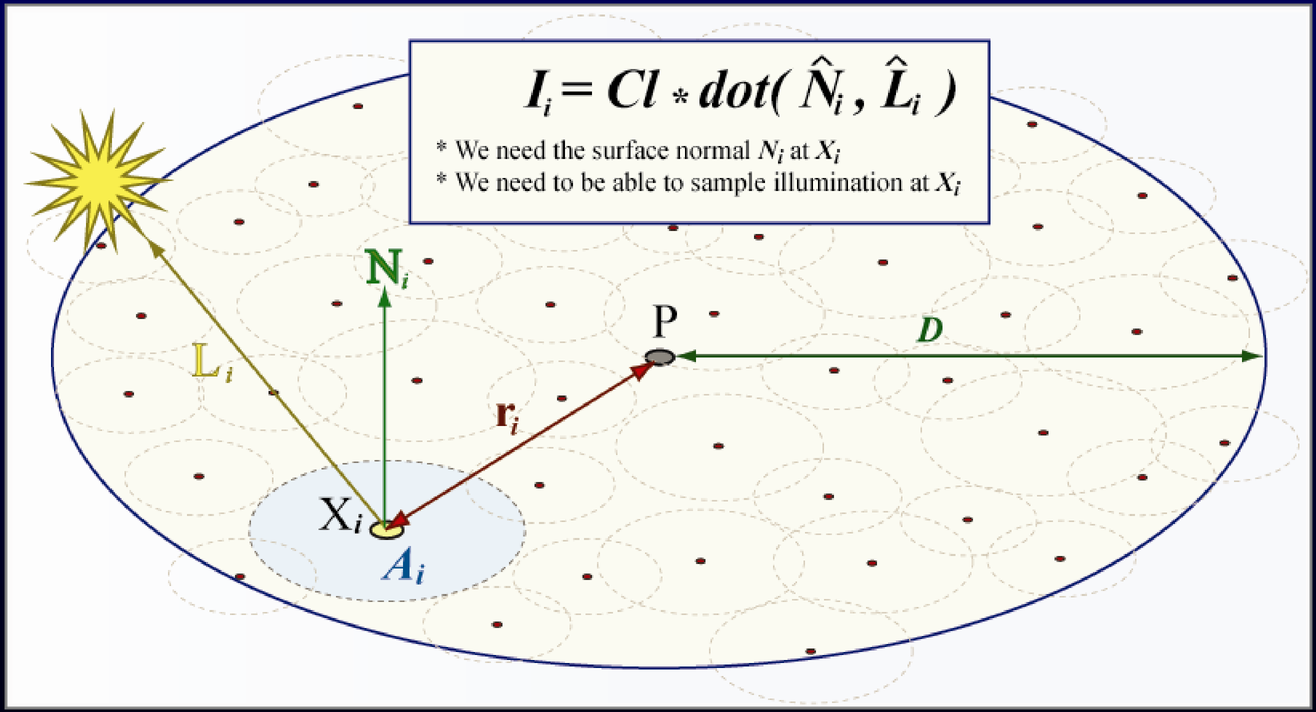
1. Start with a shading position P and a user-provided "Scattering Distance" D.
   - We know that only points within a radius D of P can possibly contribute to SSS.
2. Calculate a normalizing factor for this radius: $norm = 3\pi D^2 / 10$.
3. Distribute sample points X_i within a radius D of shade point P.
   - For each sample point X_i:
     1. Calculate representative surface area A_i.
     2. Calculate distance from P as $r_i = distance(P, X_i)$.



Sampling Strategy

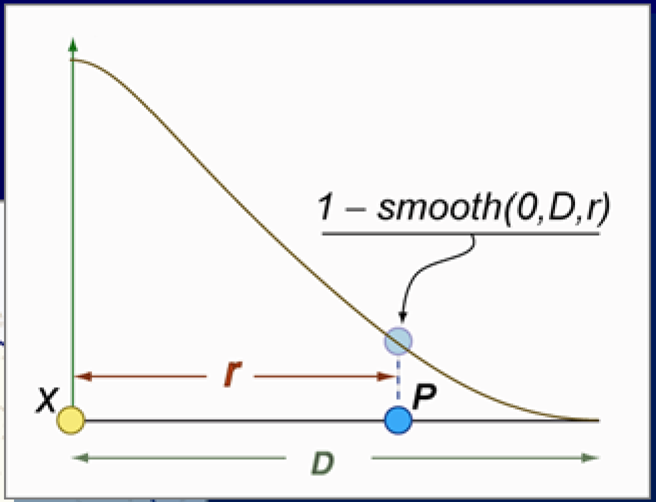
- We know that only points within a radius D of P can possibly contribute to SSS.

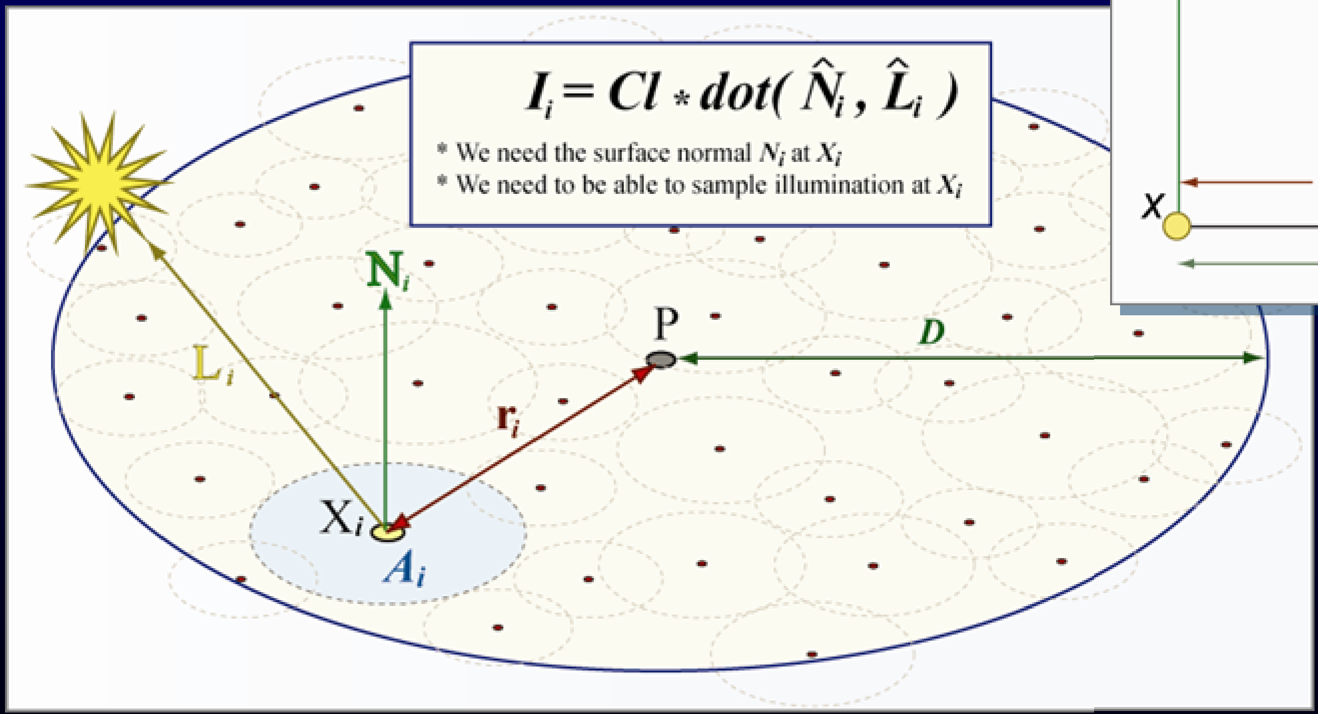
2. Calculate a normalizing factor for this radius: $norm = 3 \pi D^2 / 10$.
3. Distribute sample points X_i within a radius D of shade point P.
   - For each sample point X_i:
     1. Calculate representative surface area A_i.
     2. Calculate distance from P as $r_i = distance(P, X_i)$.
     3. Calculate irradiance $I_i = Cl * dot(N_i, L_i)$.



Sampling Strategy

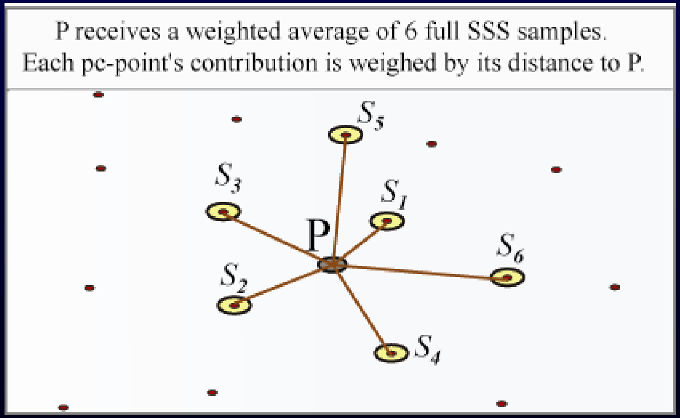
- For each sample point X_i:
  1. Calculate representative surface area A_i.
  2. Calculate distance from P as $r_i = distance(P, X_i)$.
  3. Calculate irradiance $I_i = Cl * dot(N_i, L_i)$.
  4. Calculate final contribution K_i from X_i as
     $K_i = I_i * (1\text{-}smooth(0, D, r_i)) * A_i \ / \ norm$
  5. Add K_i to accumulator.





Point Cloud Strategy

- There's no reason why we can't apply the sampling strategy to point cloud points alone.
- For every shade point:
  1. Find N neighboring pc points, where N is a parameter given by the user.
  2. Calculate SSS for each pc point in this group:
     - Store result in pc attribute
- Result for current shade point is some weighted average of all the N neighboring PC points.
- The stored SSS values for PC points used during this calculation won't need to be re-computed for nearby shade points.



A “Better” Parameterization (?)

- Instead of using a number ***N*** of points closest to ***P*** to filter over...
- We could use a filtering ***radius*** for the reconstruction.
- So we ask the user for a “Filtering Radius” instead of “Number of Points to Filter”.
- This avoids the problem of having to constantly update the number of points that will give us a “smooth” reconstruction as the cloud density changes.
- Additionally, we could internally initialize this value to some fraction of the “Scattering Distance” ***D***, and just present the user with a “Filter Size” scaling factor instead.

Attribute Shopping List

- Our PC points will need the following attributes:
  - Surface normal: ***N*** *(Facet SOP)*
  - Representative Surface Area: ***ptarea*** *(Scatter SOP)*

- Due to the way in which ptarea is calculated, we will need to “normalize” it against the total surface area.
  - Measure each prim’s area (Measure SOP).
  - Accumulate into detail attribute via the Attribute Promote SOP → Tarea.
  - Use the same process to calc total ptarea → Tptarea.
  - Final ptarea then becomes:

ptarea = ptarea * Tarea / Tptarea

What About Color?

- Two Choices:
  1. Sample unmodified irradiance and “tint” it by a surface color after the fact.
  2. Sample each “wavelength” (RGB) using different scattering distances for each one:
     - Simply use the surface color as three separate weights for the scattering distance ***D***.
     - E.g: If surface color is

       Cs = { 1, 0.5, 0 },

       and scattering distance is given as

       D = 2,

       then we will sample SSS three times (once per channel), using the scattering distances:

       D = 2, D = 1, and D = 0. *(2*1, 2*0.5, and 2*0)*
     - The normalizing factor ***norm*** should always be based on the largest value of ***D.***

Code: The Main Entry Point

```
//------------------------------------------------------------------------------
// Computes outgoing radiance due to multiple scattering at the given
// surface position, by filtering neighbouring point cloud positions.
//------------------------------------------------------------------------------

vector ssMulti (
   string   lmask;         // Light mask
   string   pcmap;         // Pointcloud map

   int      nfp;           // Number of points to filter

   vector   Rd;            // diffuse reflectance (Rd)
   float    sd;            // scattering distance (ld)

   float    bounce;        // Bounce Attenuation bias
   int      t_rgb;         // Whether to calc rgb separately

   vector   Pin;           // Surface position    [typically: P]
   vector   Nin;           // Surface normal      [typically: N]

   )
{
   vector Xo = wo_space(Pin);
   vector No = normalize(wo_nspace(Nin));
   vector mapP, mapN, ssm;
   int    xxx;

   string ch_ssm = "ssM";

   //int handle = pcopen(pcmap, "P", Xo, 1e37, nfp);
   int handle = pcopen(pcmap, "P", Xo, "N", No, 1e37, nfp);

   while (pcunshaded(handle, ch_ssm)) {
      pcimport(handle, "P", mapP);
      pcimport(handle, "N", mapN);

      ssm = ssIntegMulti ( lmask,pcmap, Rd, sd, bounce,t_rgb, mapP, mapN );

      xxx = pcexport(handle, ch_ssm, ssm);
   }

   vector bssrdf = vector(pcfilter(handle, ch_ssm));
   pcclose(handle);
   return bssrdf;

}
```

Code: The Sampling Function

```
//-----------------------------------------------------------------------------
// Integrates the multiple scattering term in the bssrdf for a single
// point-cloud point.
//-----------------------------------------------------------------------------

vector ssIntegMulti (
   string   lmask;          // Light mask
   string   pcmap;          // Pointcloud map

   vector   Rdo;            // diffuse reflectance
   float    sd;             // scattering distance

   float    bounce;         // Bounce Atten
   int      t_rgb;          // Whether to calc RGB separately

   vector   pcP;            // PointCloud position  (object space)
   vector   pcN;            // PointCloud normal    (object space)
   )
{
   vector Xi,Ni;            // For the incomming side: P,N

   vector   Xo = pcP;                   // outgoing pos
   vector   No = normalize(pcN);        // outgoing normal

   vector ld   = Rdo*sd;
   float  ld1  = max(ld);

   // Open up the point cloud map
   int   handle = pcopen(pcmap, "P", Xo, ld1, (int)1e9);

   // calc direct illumination
   pcIllum(handle,"illum",lmask);
```

Continued…

Code: The Sampling Function *(continued…)*

```
    // calc multiple scattering term
    float  r,ptarea;
    vector ssm=0, ptillum=0;
    while (pciterate(handle)) {
       pcimport(handle, "P", Xi);             // incoming pos
       pcimport(handle, "N", Ni);             // incoming normal
       pcimport(handle, "point.distance", r); // distance to Xi
       pcimport(handle, "ptarea", ptarea);    // TODO: ensure ptarea exists
       pcimport(handle, "illum",  ptillum);   // irradiance at Xi

       Ni = normalize(Ni);

       // Avoid (attenuate) light bouncing through air
       vector Li = (Xo-Xi)/ld1;               // "incidence" vector
       float  kb = ssBounceAtten(No,Ni,Li);  // bounce atten
       kb = lerp(1.0,kb,bounce);

       if(kb>0.0 ) {
          if(t_rgb)
          {
             int wave;
             for(wave=0;wave<3;wave++) {
                setcomp( ssm,
                         getcomp(ssm,wave) +
                            kb * getcomp(ptillum,wave) * ptarea *
                            (1-smooth(0,getcomp(ld,wave),r)),
                         wave
                       );
             }
          }
          else
             ssm += kb * ptillum * ptarea * (1-smooth(0,ld1,r));

       }

    }

    pcclose(handle);
    if(!t_rgb) ssm*=Rdo;

    float  norm = 3.0*ld1*ld1*A_M_PI / 10.0;
    return ssm / norm;

}
```

Code: The Irradiance Function

```
//-----------------------------------------------------------------------
// PC: Gather direct illumination (P and N are stored in object space!)
//-----------------------------------------------------------------------
void pcIllum (int handle; string att, lmask) {
   vector p, n;
   vector illum;
   int status;

   while (pcunshaded(handle, att)) {
      pcimport(handle, "P", p); p = ow_space(p);
      pcimport(handle, "N", n); n = normalize(ow_nspace(n));
      illum = 0;
      illuminance(p, n, A_M_PI_2, A_LIGHT_DIFFSPEC, "lightmask",lmask) {
         shadow(Cl);
         illum += Cl * diffuseBRDF(normalize(L), n);
      }
      status = pcexport(handle, att, illum);
   }
}
```

Next: Single Scattering…